EyeCandy3D User Manual

A 3D Scenegraph built on OpenGL

Inhaltsverzeichnis

1 Introduction

EyeCandy3D is a scene graph based on OpenGL. Every program consists of one **Application**, which holds a number of windows. Each window can hold a variable number of scenes, which can run simultaneously.

2 First steps

In this section I will guide you through the most important aspects of using this library.

2.1 Creating an application

An **Application** is the main part of this library.

```
Application app;
```

It allows the creation of new windows, which will automatically visualized. The following function will create a new window and add it to the application:

```
app.createWindow<WindowType>(...);
```

WindowType has to be an own **Window** implementation.

2.2 Creating a window

Custom windows must be derived from **ec::Window**. This window has to be instanced with the window creation function provided by an **Application**.

```
class ExampleWindow : public ec::Window
{
public:
        explicit ExampleWindow(unsigned int windowWidth,
                unsigned int windowHeight,
                const std::string& title)
        // Own window implementation ...
};

ec::Application app;
app.createWindow<ExampleWindow>(width, height, "Example Window", "id
    ");
```

The dervied window **must** support a constructor with the following signature:
MyWindow(unsigned int, unsigned int, const std::string&)

2.3 Creating a scene

Custom scenes must be derived from **ec::Scene**. By registering it with the scene manager in the associated window, it can receive updates.

```
class ExampleScene : public ec::Scene
{
public:
        explicit ExampleScene(const std::string& name,
                ec::Window* window);

        // Own window implementation ...
};

// In ExampleWindow:
void ExampleWindow::initScenes()
{
        // Create new window
        auto exampleScene = new ExampleScene("example", this);

        // Register the new window, so it receives updates
        m_sceneSystem.registerScene(exampleScene);

        // ...
}
```

2.4 Resource registry

Built in types:

-

3 Scene graph

3.1 Node

3.2 Texture

The texutre class encapsulates a texture, which resides on the gpu. There are two functions, which allow the creation of either 2D or 3D textures.

```
class Texture
{
public:
        explicit Texture();
        ~Texture();

        /**
        * \brief Create a 2D texture.
        * \return True if creation was successful, false otherwise.
        */
        bool textureFromFile(const char* path, const std::string&
           type);

        /**
        * \brief Create a 3D texture.
        * \return True if creation was successful, false otherwise.
        */
        bool cubeMapFromFile(const char* path, const std::string&
           type);
};
```

3.3 Material

A **Material** contains a number of properties, which defines the color of an object. These properties include:

- Ambient, Diffuse, Specular colors
- Shininess factor for highlights
- Textures

3.4 Geometry

A geometry object encapsulates various geometry data, including:

- Vertices
- Normals
- Texture coordinates

It is responsible for memory managment on the GPU regarding geometry data.

There already are specific geometry implementations for common geometry types:

3D :

- CubeGeometry
- SphereGeometry
- CylinderGeometry

2D :

- CircleGeometry
- RectangleGeometry

If these predefined types aren't sufficicent, you can create your own geometry types. This is done by deriving from **StaticGeometry**.

3.5 Shader

The **Shader** class can load, compile and link shader programs. The shading language used is GLSL, since OpenGL is being used.

3.6 Drawable

A drawable groups together

- Geometry,
- Material and
- Shader

Drawables can be added to scene graph **Node**s so they are rendered.

4 Rendering

4.1 Frame

4.2 Camera

5 Lighting

TBA

6 Input

Input is provided through the GLFW library.

6.1 Input events

Input events are always bound to one window. There are multiple sources, which can generate input events:

- **Mouse**
- **Keyboard**
- **Window**
- **Joystick**

An **InputEvent** consists of an **InputType** and the **EventData**. The **EventData** holds all different kinds of events, of which only one can be active at a time due to it being a union. The **InputType** describes, which part of the **EventData** is active. All other elements inside **EventData** are invalid!

I.e. if the **InputType** is *key_pressed*, only the KeyboardEvent inside the **EventData** is active.

6.2 EventSystem

An event system is always linked to exactly one window.

DeviceRegistry:
Contains input devices (mouse, keyboard, joystick etc.). Those devices can be activated to generate input events.

InputObservable:
InputListener can be registered at this component, which will then be notified about incoming **InputEvents**.

InputListener:
Input listener contain a number of callbacks, which can be added by the user. Input listener have to be registered at an **InputObservable** located in a window, to be informed about input events.

7 GUI

The GUI is built using the Agui-library with custom backends for OpenGL.

7.1 Initialization

Before the GUI library-part can be used, it has to be initialized. To do so, use the static function:

```
void ec::MiniAgui::init();
```

7.2 Backend

7.3 title

Abbildungsverzeichnis

Tabellenverzeichnis